Extending dataflow programs with throughput properties

Manuel Selva
Bull Echirolles, 1 rue de Provence 38432 Echirolles Cedex, France
manuel.selva@ext.bull.net

Lionel Morel
Université de Lyon INSA-Lyon, CITI-INRIA F-69621 Villeurbanne, France
lionel.morel@insa-lyon.fr

Kevin Marquet
Université de Lyon INSA-Lyon, CITI-INRIA F-69621 Villeurbanne, France
kevin.marquet@insa-lyon.fr

Stéphane Frénot
Université de Lyon, INRIA INSA-Lyon, CITI-INRIA, F-69621 Villeurbanne, France
stephane.frenot@insa-lyon.fr

ABSTRACT

In the context of multi-core processors and the trend toward many-core, dataflow programming can be used as a solution to the parallelization problem. By decoupling computation from communication, this paradigm naturally exposes parallelism in several ways. In this work we propose language extensions for expressing throughput properties over dataflow programs together with a run-time mechanism for the observation of events meaningful to compute the effective throughput. We show the limited impact of such mechanisms on the application overall performances. We also review existing run-time adaptation mechanisms that may be used in a dataflow context to satisfy throughput requirements.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; D.3.2 [**Programming Languages**]: Language Classifications—*Data-flow languages*

Keywords

Dataflow, Quality of Service, Monitoring, Throughput

1. INTRODUCTION

The trend toward more parallelization in general-purpose computers started in 2005 and is now leading to many-core processors [4]. The software industry is facing challenging concerns about programming such hardware. The question of which programming model to adopt in this context remains open. The imperative concurrent programming model *a-la* Pthread is today prevalent in the industry even if strong arguments go against its use [9]. Particular classes of programs may benefit from other programming models, both from the expressiveness and the optimization points of view. Our work focuses on the dataflow programming paradigm described below and on applications with quality-of-service requirements. These include audio or image processing, trading applications, telecommunication data processing, stream compression and encryption.

In this paper, we first show how dataflow languages can be extended to express an expected throughput. We then detail how dataflow compilers are modified to generate monitoring code verifying that the throughput expressed by the programmer is respected at run-time.

MES '13, June 23 - 24 2013, Tel-Aviv, Israel

2. EXECUTING DATAFLOW PROGRAMS

We consider the dataflow programming model introduced by Lee and Parks in [11] and one of its specialization called Synchronous Dataflow [10] (SDF). We recall the foundation of these models before introducing how dataflow programs are transformed to be executed on the execution model we target.

2.1 Dataflow programming

A dataflow program is a graph of actors. An actor represents a computation unit. Actors are connected with each other through communication channels. They consume tokens on their input channels and produce tokens onto their output channels. Writing a token adds it at the end of the channel and reading returns a single token (if available) in a First-In-First-Out (FIFO) manner. At a given time, a channel contains all the tokens produced by the writing actor but not yet consumed by the reading actor. These FIFO channels are the only way through which actors can communicate, which makes this a functional model well-suited for compositional reasoning. Actors and channels form dataflow networks that can be hierarchically composed to form complex systems.

The execution of an actor is driven by the availability of tokens on its input channels. From an external point of view, an actor can perform two operations: reading tokens from its inputs channels and writing tokens to its output channels. This defines the model of communication of programs.

The complementary model of computation expresses how new values are computed before they are communicated on output channels. We do not detail it here.

In the SDF model, the number of tokens needed by an actor is known at compile time. This allows to statically compute a schedule of the actors and to bound the FIFO queues size.

Fig. 1 illustrates actors and channels concepts with an MPEG 4 part-2 simple profile decoder application. This example explicits the different kind of parallelism exposed by the dataflow programming model. Luminance (Y) and chrominances (U,V) parallel decoding is tasks parallelism. Pipeline parallelism is achieved by letting consumers working ahead of producers. Finally, data parallelism concerns the duplication of actors without state such as `Text Y` in this example.

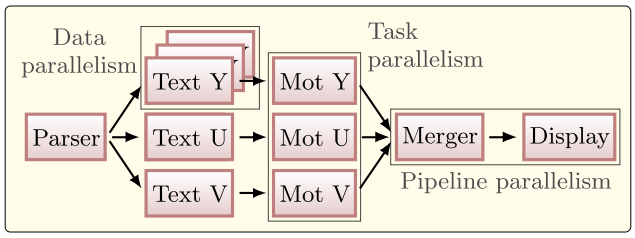


Figure 1: MPEG 4 part-2 decoder dataflow graph

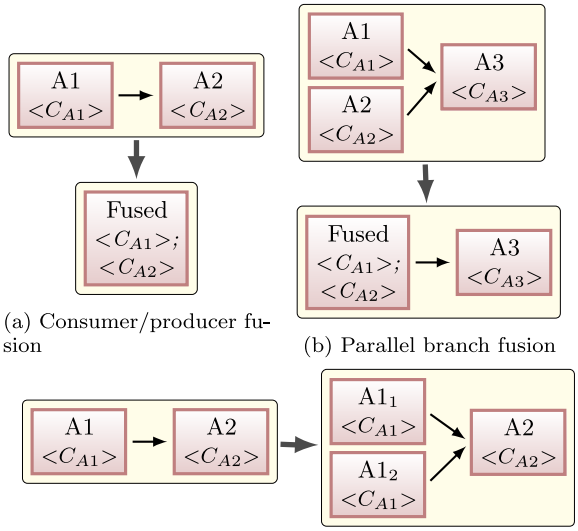
2.2 Compilation and execution model

We focus on dataflow programs executed using a set of imperative concurrent tasks. These tasks run concurrently on one or more hardware execution units. Communication mechanisms, e.g. shared memory or inter-core event mechanisms, are used to let tasks exchange data and to synchronize. Targeting this execution model, a compiler transforms a dataflow program into a set of imperative concurrent tasks.

Depending on the underlying hardware, task execution model implementations may vary. Dataflow compilers targeting standard multi-core architectures such as [13, 3] generate C or C++ code using Pthreads. These files need then to be compiled to machine code. The communication between threads is implemented using global variables and POSIX synchronization mechanisms. Other dataflow compilers are dedicated to specific many-core hardware [7, 6].

Dataflow compilers perform optimizations to exploit parallelism. As reported in [6], these compilers often have to perform complex transformations on the initial dataflow graph to extract the right degree of parallelism for a specific hardware target.

The first kind of transformations consists in fusing actors. They are used for example when the number of actors is larger than the number of execution units dedicated to the application. Fusions can either merge consumer/producer pairs or parallel branches of actors into a single actor as shown on Fig. 2a and Fig. 2b. In both cases, the compiler puts in sequence in a new actor, producer's code respectively first branch actor's code with consumer's code respectively second branch actor's code. The second class of transformation adds parallelism to the dataflow graph. Data parallelism introduction depicted on Fig. 2c duplicates stateless actors to allow them to work on several data sets in parallel. Pipeline parallelism introduction splits actors into a pipeline of smaller actors.



(a) Consumer/producer fusion

(b) Parallel branch fusion

(c) Data parallelism introduction

Figure 2: Compilers transformations - Ax are actors names and <C_{Ax}> represent the sequential code of actors.

3. EXTENDING DATAFLOW PROGRAMS

We now describe our approach for expressing expected throughput, observing applications and building effective throughput information.

3.1 Expressing throughput properties

The Expected ThroughPut (ETP), is expressed at application's source level on the dataflow graph. This can be done either on channels or inside actors. On a channel, it represents the number of tokens that must enter the channel every time unit. Inside an actor, a throughput property needs to be defined by the programmer, e.g. as the number of times a given atomic action is performed every time unit. This is too intrusive and we shall prefer expressing throughput objectives on channels. In Fig. 3, the channel connecting the `Merger` and `Display` actors is tagged with a throughput information $ETP = 25$, expressed in number of frames per second.

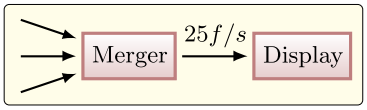
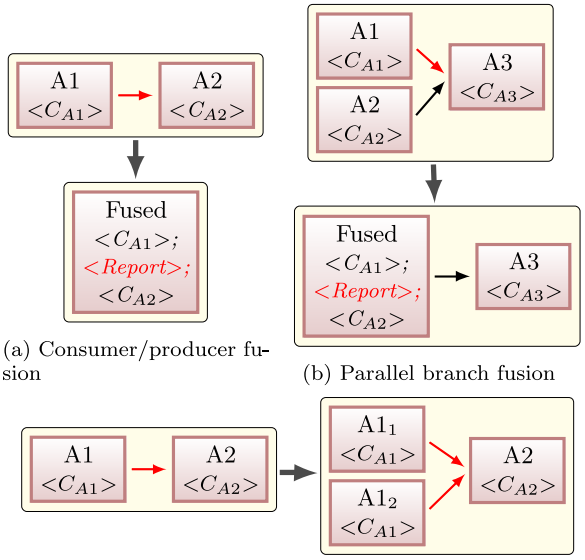


Figure 3: Mpeg example with throughput information.

3.2 Reporting

The reporting mechanism is responsible for gathering and storing information allowing to compute the effective throughput, denoted OTP for Observed ThroughPut. The information we report is the number of tokens written on a given monitored channel since the application's start time. This information must be stored in a location where it can be shared by the tasks implementing actors and the ones implementing the monitoring and decision mechanisms described in section 3.3.

To implement the reporting, we adapt the compiler to generate reporting code. The token count is initialized to



(a) Consumer/producer fusion

(b) Parallel branch fusion

(c) Data parallelism introduction

Figure 4: Impact of transformations on reporting - Red arrows are monitored channels, Ax are actors names, $<C_{Ax}>$ represent the sequential code of actors and <*Report*> are reporting annotations.

zero when the application starts. Then, for each token written on an observed channel, the instrumented version of the actor increments the token counter with the number of written tokens. There is no need to reset counters because the monitor uses counters differences to compute the OTP.

The process of adding reporting code when a token on a monitored channel is written is straightforward for compilers who don't apply transformations. For compilers applying the transformations described in section 2.2, we need to follow monitored channels across these transformations. Fig. 4a and Fig. 4b illustrate how reporting is handled is the case of fusion. To guaranty the reporting as initially expressed, we keep track in the fused actor of the frontier between the two initial actors. Fig. 4c shows how data parallelism introduction impacts reporting . When one of the initial actor's output channel is monitored, we add reporting on each created duplicate. In the case of pipeline parallelism introduction, we report the monitoring of the split actor on the last actor of the created pipeline.

3.3 Monitoring

To compute the OTP, the monitor performs simple arithmetic on token counts provided by the reporting system. The comparison of this OTP and the ETP is done at regular time intervals according to the monitoring frequency. When the OTP becomes lower than the ETP, the decision making mechanism is invoked.

In the targeted execution model, we implement this monitor in one or more dedicated tasks for each monitored application. The number of monitoring tasks and their location is an important parameter regarding the introduced overhead. When enough free execution units exists, they are naturally used to execute the monitoring. When the number of monitoring tasks is greater than the number of free execution units, it is strongly recommended to locate monitoring tasks on execution units with the lowest load.

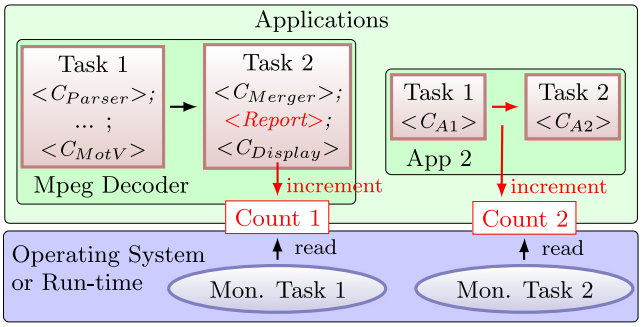


Figure 5: System overview

Another important parameter is the frequency of the monitoring because of its potential impact on performances, the higher the frequency, the bigger the overhead. It can be configured independently for each monitored channel. The maximum pertinent frequency is directly linked to the expected throughput rate: in the case of a throughput of 25 frames per second, monitoring at a frequency of 25Hz is enough. Indeed, the decoder never sends more than 1 frame every $1/25s$. On the other hand a low frequency will decrease the reaction time of our run-time mechanisms. The value for this parameter is strongly application and context dependent.

[1] provides a complete report with overhead evaluation for different frequencies, different numbers and different locations of the monitoring tasks.

3.4 System overview

Fig. 5 illustrates the mechanisms described so far. In this scenario the mpeg decoder of Fig. 1 is running along with another dataflow application called App 2. Both applications have been compiled to two tasks and have exactly one monitored channel. The monitored channel of the mpeg decoder is located in a task resulting of actors fusion. App 2 has not been transformed at all and its monitored channel still exists at run-time. In this scenario we arbitrarily decide to create one monitoring task for each monitored channel and we can see how monitoring tasks gets reporting information: applications increments tokens counts shared with monitoring tasks.

3.5 Toward dynamic adaptations

If the OTP is lower than the ETP, dynamic adaptations must be performed. It is beyond the scope of this article to detail which dynamic adaptations to use ; we focus on detecting unacceptable throughput. However, we are already investigating dynamic adaptations mechanisms. We consider how the dataflow graph imperative task mapping can be reconsidered at runtime. Existing work in this direction [12] proposes to statically specify different dataflow graphs for the same application. These graphs are assigned with QoS information. Depending on the observed performances the run-time system can then choose among these configurations. We are also working on reconfiguration mechanisms such as the deactivation and reactivation of optional actors and large actors internal reconfiguration. This last adaptation requires actors to provide quality selection parameters. Finally, reconfiguration of the overall dataflow graph can be performed.

4. EXPERIMENTS

The concepts previously introduced have been validated into the StreamIt [13] language rooted in the SDF model. We used the compiler back-end generating C source code using Pthreads to evaluate the impact of our mechanisms on a standard multi-core platform running Linux. See [1] for implementation details and overhead evaluation.

5. RELATED WORKS

Different execution layers for dataflow programs have been set up [5], even for multi-core machines [3, 14]. However, none of these is explicitly concerned with dynamic optimizations nor the respect of QoS properties. [2] tackles dynamism in an ad-hoc manner while we try to be as generic as possible. To our knowledge , the work described in [12] and the Flextream [8] approach are the closest to our proposition, each taking a route that is rather orthogonal to the other one. The first approach considers a set of pre-defined, statically built, configurations of dataflow graphs and dynamically decides, depending on observed conditions over the whole system's performances, which configuration to apply at given switch points in time. On the other hand, Flextream does not supposes any pre-defined configurations instead allowing for dynamically recomputing partitioning of actors among available computing units in order to meet run-time performance constraints.

6. CONCLUSION

We believe that dataflow is an attractive alternative to write efficient parallel program. Parallelism is clearly exposed in this programming model and can be one of the answer to the question on how program tomorrow massively parallel architectures. Dynamic adaptation of dataflow programs is a hot topic for ensuring the satisfaction of quality-of-service requirements of applications at run-time. In this paper, we present a solution allowing to monitor throughput properties of dataflow programs with QoS requirements. The solution has been evaluated on a general purpose multi-core platform showing the limited overhead of the solution.

In this work, we ran our monitoring mechanism together with only one dataflow application. In the short term, we intend to adapt it so that it can monitor different applications. Moreover, the possibility to attach ETPs to different channels in dataflow graphs allows to imagine dynamic adaptations at the granularity of actors and not only of applications.

We want to conduct comparable studies on different dataflow languages and architectures. The next step in that direction will be to adapt our proposal to languages such as ΣC [7] targeting many-cores.

Finally, we plan to deeply investigate adaptation mechanisms. We want to study how information exposed by the dataflow programming model could help in applying well-known run-time adaptation mechanisms such as load balancing, exploitation of cache affinity and affinity scheduling.

We also want to conduct a general study on the actor activation and deactivation idea. Unplugging an actor is quite simple but depending on the considered dataflow model of computation, plugging it back could be more difficult. We plan to clearly identify the required conditions for each dataflow model of computation to safely plug and unplug actors.

7. REFERENCES

[1] `http://perso.citi-lab.fr/mselva/mes13/dataflow-experiments.pdf`.

[2] A. Albers and P. With. Task complexity analysis and qos management for mapping dynamic video-processing tasks on a multi-core platform. *Journal of Real-Time Image Processing*, 7:185–202, 2012.

[3] I. Amer, C. Lucarz, G. Roquier, M. Mattavelli, M. Raulet, J.-F. Nezan, and O. Deforges. Reconfigurable video coding on multicore. *Signal Processing Magazine, IEEE*, 26(6):113 –123, november 2009.

[4] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, Oct. 2009.

[5] J. Boutellier, V. Sadhanala, C. Lucarz, P. Brisk, and M. Mattavelli. Scheduling of dataflow models within the reconfigurable video coding framework. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 182–187, 2008.

[6] M. I. Gordon, D. Maze, S. Amarasinghe, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. a. Lamb, C. Leger, J. Wong, and H. Hoffmann. A stream compiler for communication-exposed architectures. *ACM SIGARCH Computer Architecture News*, 30(5):291, Oct. 2002.

[7] T. Goubier, R. Sirdey, S. Louise, and V. David. σc: A programming model and language for embedded manycores. In *Algorithms and Architectures for Parallel Processing*, volume 7016 of *Lecture Notes in Computer Science*, pages 385–394. 2011.

[8] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 214–223, 2009.

[9] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[10] E. A. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, 1987.

[11] E. A. Lee and T. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773 –801, may 1995.

[12] S. Stuijk, M. Geilen, and T. Basten. A predictable multiprocessor design flow for streaming applications with dynamic behaviour. In *Proceedings of the 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, DSD '10, 2010.

[13] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, 2002.

[14] H. Yviquel, E. Casseau, M. Wipliez, and M. Raulet. Efficient multicore scheduling of dataflow process networks. In *Signal Processing Systems (SiPS), 2011 IEEE Workshop on*, pages 198 –203, oct. 2011.